决策树分类与贝叶斯网络推理及其可视化

2050289 朱昀玮，2050283 陆明奇，1951112 林日中

摘 要

决策树是一种非参数的有监督学习方法，它能够从一系列有特征和标签的数据中总结出决策规则，并用树状图的结构来呈现这些规则，以解决分类和回归问题。决策树中每个内部节点表示一个属性上的判断，每个分支代表一个判断结果的输出，最后每个叶节点代表一种分类结果。本实验基于 Python 语言，运用 ID3，C4.5，CART 三种不同算法构建决策树进行目标分类，利用前后剪枝对决策树进行优化，并比较分析三种算法的特点及其效率，最后通过 Python 中的 PySide6 框架实现可视化。

贝叶斯网络又称信度网络，是 Bayes 方法的扩展，是目前不确定知识表达和推理领域最有效的理论模型之一。贝叶斯网络常用于确定性推理的应用中，本应用中实现了贝叶斯网络的精确推理，将该算法应用于后验概率的查询。并将两者整合在一个 GUI 框架中。

关键词：决策树，贝叶斯网络，目标分类，剪枝，Qt,Python

Classification algorithm based on decision tree and its visualization

ABSTRACT

Decision tree is a nonparametric supervised learning method. It can summarize decision rules from a series of data with characteristics and labels, and present these rules with the tree structure to solve the classification and regression problems. Each internal node in the decision tree represents a judgment on an attribute, each branch represents the output of a judgment result, and finally each leaf node represents a classification result. Based on the python language, this experiment uses ID3, C4.5 and CART to build a decision tree for target classification, uses pre-pruning and post-pruning to optimize the decision tree, compares and analyzes the characteristics and efficiency of the three algorithms, and finally realizes visualization through PySide GUI framework in Python.

Key words： decision tree, target classification, pruning,Qt,Python

目录

1 实验概述

1.1 实验目的

熟悉和掌握目标分类问题的定义及其主流求解方法，并利用决策树算法求解目标分类问题，理解求解流程和决策顺序。使用任意语言制作相关图形化界面，可视化决策树算法的决策过程、决策结果与决策树结构。

1.2 实验内容

1. 采用 Python 语言实现基于决策树的目标分类的程序，设计了三种不同的决策树算法： 第一种：ID3,使用信息增益来选择划分属性；第二种：C4.5，使用信息增益率来选择划分属性；第三种：CART,使用“基尼指数”来选择划分属性。

2. 利用预剪枝和后剪枝，对决策树进行优化，避免决策树学习过程中的过拟合问题。

3. 使用 Pyside6 库实现整体交互设计，包括训练数据和测试数据从文件导入，决策树可视化，基于测试数据的运行结果显示等。

2 实验方案设计

2.1 总体设计思路与总体架构

程序整体采用求解与图形化界面分离的策略，分别在 tree.py 与 xxx.py 中实现决策树算法求解目标分类问题与 GUI 图形化界面。

2.2 核心算法及基本原理

决策树是一种描述对实例进行分类的树形结构。决策树由点和有向边组成。节点有两种类型：内部节点和叶节点。内部节点表示一种特征或者属性，叶节点表示一个分类。构建决策树时通常采用自上而下的方法，在每一步选择一个最好的属性来分裂。

“最好”的定义是使得子节点中的训练集尽量的纯。不同的算法使用不同的指标来定义“最好”。决策树构建过程中，首先构建根节点，将所有训练数据放在根节点，选择一个最优特征，按照这一特征的取值将训练数据分割为子集，使各个子集有一个当前条件下最好的分类。如果这些子集能被基本正确分类，那么构造叶节点，将对应子集集中到叶节点。如果有子集不能被正确分类，那么就这些子集选择新的最优特征，继续对其进行分割，构建相应的节点。

递归进行上述的操作，直到所有训练数据子集均能被正确分类。决策树每次都找不同的切分点，将样本空间逐渐进行细分，最后把属于同一类的空间进行合并，就形成了决策边界，树的层次越深，决策边界的切分就越细，区分越准确，同时也越有可能产生过拟合。本实验采用三种不同的属性划分标准来构建决策树。

算法	树结构	支持模型	特征选择	连续值处理	缺失值处理	剪枝
ID3	多叉树	分类	信息增益	不支持	不支持	不支持
C4.5	多叉树	分类	信息增益率	支持	支持	支持
CART	二叉树	分类、回归	基尼系数、平方误差和	支持	支持	支持

2.2.1 ID3 算法

ID3 算法是在决策树的各个结点上应用信息增益准则进行特征选择。

特征选择也即选择最优划分属性，从当前数据的特征中选择一个特征作为当前节点的划分标准。随着划分过程不断进行，希望决策树的分支节点所包含的样本尽可能属于同一类别，即节点的“纯度”越来越高。

熵表示事务不确定性的程度，也就是信息量的大小，可以表示为

$$\text{Entropy} = -\sum_{i=1}^{n} p(x_i) * \log_2 p(x_i)$$

其中，$p(x_i)$是分类x_i出现的概率，n是分类的数目。可以看出，熵的大小只和变量的概率分布有关。

对于在X条件下Y的条件熵，有

$$Entropy(Y|X) = \sum_{i=1}^{n} p(x_i) Entropy(Y|x_i)$$

所以当$Entropy$最大为 1 的时候，是分类效果最差的状态；当它最小为 0 的时候，是完全分类的状态。熵等于 0 是理想状态，一般实际情况下，熵介于 0 和 1 之间 。

熵的不断最小化，实际上就是提高分类正确率的过程。

在划分数据集之前之后信息发生的变化，计算每个特征值划分数据集获得的信息增益，获得信息增益最高的特征就是最好的选择。

我们定义属性A对数据集D的信息增益为$infoGain(D|A)$，它等于D本身的熵，减去给定A的条件下D的条件熵，即

$$infoGain(D|A) = Entropy(D) - Entropy(D|A)$$

这样一来，计算每个属性引入后的信息增益，选择给D带来的信息增益最大的属性，即为最优划分属性。一般，信息增益越大，则意味着使用属性A来进行划分所得到的的“纯度提升”越大。

ID3 算法的具体做法是：

a. 从根节点开始，对结点计算所有可能特征的信息增益，选择信息增益最大的特征作为结点的特征，并由该特征的不同取值构建子节点；

b. 对子节点递归地调用以上方法，构建决策树；

c. 直到所有特征的信息增益均很小或者没有特征可选时为止。

它的主要优点是，计算复杂度不高，输出结果易于理解，对中间值的缺失不敏感，可以处理不相关特征数据；主要缺点是，**没有剪枝**，可能会产生过度匹配问题，需要进行剪枝，且采用信息增益作为选择最优划分特征的标准，然而**信息增益会偏向那些取值较多的特征**。

2.2.2 C4.5 **算法**

信息增益准则对可取值数目较多的属性有所偏好，为减少这种偏好可能带来的不利影响，C4.5 算法采用信息增益率来选择最优划分属性。即

$$GainRatio(D|A) = \frac{infoGain(D|A)}{IV(A)}$$

$$IV(A) = -\sum_{k=1}^{K} \frac{|D_k|}{|D|} * log_2 \frac{|D_k|}{|D|}$$

其中$A = [a_1, a_2, \cdots, a_k]$包含了 K 个值。若使用$A$来对样本集$D$进行划分，则会产生$K$个分支节点，其中第$k$个节点包含$D$中所有属性$A$上取值为$a_k$的样本，记为$D_k$。通常，属性$A$的可能取值数越多（即$K$越大），则$IV(A)$的值通常会越大。

信息增益率准则对可取值数目较少的属性有所偏好。所以，C4.5 算法不是直接选择信息增益率最大的候选划分属性，而是先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择信息增益率最高的。

当属性类型为离散型，无须对数据进行离散化处理；当属性类型为连续型，则需要对数据进

行离散化处理。具体思路如下：

a. m个样本的连续特征A有m个值，从小到大排列$a_1, a_2, \cdots, a_m$，取相邻两样本值的平均数做划分点，一共有$m-1$个，其中第i个划分点T_i表示为：$T_i = (a_i + a_{i+1})/2$。

b. 分别计算以这$m-1$个点作为二元切分点时的信息增益率。选择信息增益率最大的点为该连续特征的最佳切分点。例如，取到的信息增益率最大的点为a_t，则小于a_t的值为类别 1，大于a_t的值为类别 2，这样就做到了连续特征的离散化。

相比 ID3 算法，C4.5 算法更换了特征选择的标准，使用**信息增益比**进行特征选择。不直接选择增益率最大的候选划分属性，候选划分属性中找出信息增益高于平均水平的属性，这样保证了大部分好的的特征。再从中选择增益率最高的，这又保证了不会出现编号特征这种极端的情况。

它的主要优点是，产生的分类规则易于理解，准确率较高；主要缺点有：

- C4.5 算法只能用于分类；
- C4.5 是多叉树，用二叉树效率会提高；
- 在构造树的过程中，需要对数据集进行多次的顺序扫描和排序（尤其是对连续特征），因而导致算法的低效；
- 在选择分裂属性时没有考虑到条件属性间的相关性，只计算数据集中每一个条件属性与决策属性之间的期望信息，有可能影响到属性选择的正确性；
- C4.5 只适合于能够驻留于内存的数据集，当训练集大得无法在内存容纳时程序无法运行。

2.2.3 CART 算法

CART 算法即分类回归树算法（Classification and Regression Tree，CART）。ID3 和 C4.5 虽然在对训练样本集的学习中可以尽可能多地挖掘信息，但是其生成的决策树分支、规模都比较大，CART 算法的二分法可以简化决策树的规模，提高生成决策树的效率。

数据集D的纯度可用基尼值来度量：

$$Gini(D) = \sum_{i=1}^{n} p(x_i) * \left(1 - p(x_i)\right)$$
$$= 1 - \sum_{i=1}^{n} p^2(\mathrm{x_i})$$

其中，$p(x_i)$是分类x_i出现的概率，n是分类的数目。$Gini(D)$反映了从数据集D中随机抽取两个样本，其类别标记不一致的概率。因此，$Gini(D)$越小，则数据集D的纯度越高。

对于样本D，个数为$|D|$，根据特征A是否取某一可能值a，把样本D分成两部分D_1, D_2。也即

$$D_1 = \{(x, y) \in D | A(x) = a\}, D_2 = D - D_1$$

所以 CART 分类树算法建立起来的是二叉树，而不是多叉树。

在属性A的条件下，样本D的基尼系数定义为

$$GiniIndex(D|A = a) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

与 C4.5 思想相同，都是将连续的特征离散化。区别在选择划分点时，C4.5 是信息增益率，CART 是基尼系数。

c. m个样本的连续特征A有m个值，从小到大排列$a_1, a_2, \cdots, a_m$，则 CART 取相邻两样本值

的平均数做划分点，一共有$m-1$个，其中第i个划分点T_i表示为：$T_i=(a_i+a_{i+1})/2$。

d. 分别计算以这$m-1$个点作为二元分类点时的基尼系数。选择基尼系数最小的点为该连续特征的二元离散分类点。比如取到的基尼系数最小的点a_t，则小于a_t的值为类别 1，大于a_t的值为类别 2，这样就做到了连续特征的离散化。

需要注意的是，与 ID3、C4.5 处理离散属性不同的是，如果当前节点为连续属性，则该属性在后面还可以参与子节点的产生选择过程。

CART 算法按如下步骤进行建树。

输入：训练集D，基尼系数的阈值，切分的最少样本个数阈值

输出：分类树T

从根节点开始，用训练集递归建立 CART 分类树。

a. 对于当前节点的数据集为D，如果样本个数小于阈值或没有特征，则返回决策子树，当前节点停止递归；

b. 计算样本集D的基尼系数，如果基尼系数小于阈值，则返回决策树子树，当前节点停止递归 ；

c. 计算当前节点现有各个特征的各个值的基尼指数，

d. 在计算出来的各个特征的各个值的基尼系数中，选择基尼系数最小的特征A及其对应的取值a作为最优特征和最优切分点。然后根据最优特征和最优切分点，将本节点的数据集划分成两部分D_1和D_2，同时生成当前节点的两个子节点，左节点的数据集为D_1，右节点的数据集为D_2。

e. 对左右的子节点递归调用 a-d 步，生成 CART 分类树；

对生成的 CART 分类树做预测时，假如测试集里的样本落到了某个叶子节点，而该节点里有多个训练样本。则该测试样本的类别为这个叶子节点里概率最大的类别。

虽然 CART 是所用到的 3 种算法中最优秀的，但它还有很多不足：

a. 无论是 ID3, C4.5 还是 CART，在做特征选择的时候都是选择最优的一个特征来做分类决策，但是大多数，分类决策不应该是由某一个特征决定的，而是应该由一组特征决定的。这样决策得到的决策树更加准确，被称为多变量决策树。在选择最优特征的时候，多变量决策树不是选择某一个最优特征，而是选择最优的一个特征线性组合来做决策。

b. 如果样本发生一点点的改动，就会导致树结构的剧烈改变。

2.2.4 剪枝

前面的算法生成的决策树非常得详细而庞大，每个属性都被详细地加以考虑，决策树的树叶节点所覆盖的训练样本都是“纯”的。因此用这个决策树来对训练样本进行分类的话，它可以完美正确地对训练样本集中的样本进行分类。但是，如果训练样本中包含了一些错误，按照前面的算法，这些错误也会完全被决策树学习，导致“过拟合”。为了避免决策树“过拟合”样本，可以采用剪枝来对决策树进行优化。

预剪枝是在树的构建过程，设置一个阈值，使得当在当前分裂节点中分裂前和分裂后的误差超过这个阈值则分裂，否则不进行分裂操作。所有决策树的构建方法，都是在无法进一步降低熵的情况下才会停止创建分支的过程，为了避免过拟合，可以设定一个阈值，熵减小的数量小于这

个阈值，即使还可以继续降低熵，也停止继续创建分支。预剪枝就是在完全正确分类训练集之前，较早地停止树的生长。具体在什么时候停止决策树的生长有多种不同的方法:

a. 在决策树到达一定高度的情况下就停止树的生长。

b. 到达此结点的实例具有相同的特征向量，而不必一定属于同一类，也可停止生长。

c. 到达此结点的实例个数小于某一个阈值也可停止树的生长。

d. 计算每次扩张对系统性能的增益，如果这个增益值小于某个阈值则不进行扩展。

优点：快速，可以在构建决策树时进行剪枝，显著降低了过拟合风险。由于预剪枝不必生成整棵决策树，且算法相对简单，效率很高，适合解决大规模问题。

缺点：预剪枝基于贪心思想，本质上禁止分支展开，给决策树带来了欠拟合的风险。在相同的标准下，也许当前的扩展会造成过度拟合训练数据，但是更进一步的扩展能够满足要求，也有可能准确地拟合训练数据。这将使得算法过早地停止决策树的构造。

后剪枝是在决策树构造完成后进行剪枝。剪枝的过程是对拥有同样父节点的一组节点进行检查，判断如果将其合并，熵的增加量是否小于某一阈值。如果确实小，则这一组节点可以合并一个节点，其中包含了所有可能的结果。后剪枝删除了一些子树，并用其叶子节点代替，这个叶子节点所标识的类别通过这棵子树中大多数训练样本所属的类别来确定。相比于预剪枝，后剪枝方法更常用，因为在预剪枝方法中精确地估计何时停止树的增长非常困难。

优点：后剪枝决策树通常比预剪枝决策树保留了更多分枝，欠拟合风险小，泛化性能好。

缺点：后剪枝在生成完全决策树之后进行，自底向上对所有非叶节点进行逐一考察，训练的时间开销比未剪枝决策树和预剪枝决策树都要大得多。

2.2.5 贝叶斯网络精确推理算法

本程序中实现了贝叶斯网络精确推理的变量消元算法。变量消元算法按照从右到左的次序计算，并将中间计算结果存储下来。

$$P(b \mid j,m) = \alpha P(b)\sum_{e} P(e)\sum_{a} P(a \mid b,e)P(j \mid a)P(m \mid a)$$

在算法中，需要实现一种特殊乘法，即逐点相乘。在计算过程中，对于没有在查询的条件和变量中出现的元素，进行 sum-out 求和操作，对于在条件中的变量（j 和 m），只需要保留其特定取值即可。而对于查询的变量 B，即需要保存 B，也需要保存~B.

逐点相乘的计算方式如下所示。

A	B	$\mathbf{f}_1(A,B)$	B	C	$\mathbf{f}_2(B,C)$	A	B	C	$\mathbf{f}_3(A,B,C)$
T	T	0.3	T	T	0.2	T	T	T	$0.3 \times 0.2 = 0.06$
T	F	0.7	T	F	0.8	T	T	F	$0.3 \times 0.8 = 0.24$
F	T	0.9	F	T	0.6	T	F	T	$0.7 \times 0.6 = 0.42$
F	F	0.1	F	F	0.4	T	F	F	$0.7 \times 0.4 = 0.28$
						F	T	T	$0.9 \times 0.2 = 0.18$
						F	T	F	$0.9 \times 0.8 = 0.72$
						F	F	T	$0.1 \times 0.6 = 0.06$
						F	F	F	$0.1 \times 0.4 = 0.04$

图 14.10 逐点相乘示例：$\mathbf{f}_1(A,B) \times \mathbf{f}_2(B,C) = \mathbf{f}_3(A,B,C)$

算法的伪代码如下所示。

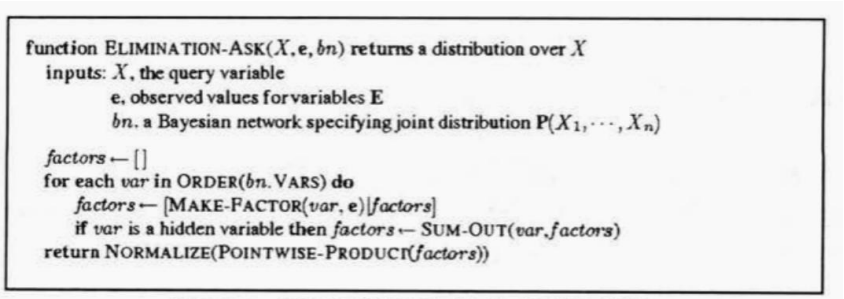


图 14.11 用于贝叶斯网络推理的变量消元算法

该算法首先对所有变量排拓扑序，从拓扑序的低处向拓扑序高位的变量逐次进行运算。然后，对每个排完序后的变量进行逐点乘法，如果某个变量为隐变量，则进行额外的 sum-out 求和操作。

算法程序描述如下。

逐点乘法

```
def mult(self,f):
        newVars = list(set(self.vars + f.vars))
        width = 2 ** (len(newVars))
        newData = np.zeros((width,len(newVars)+1))
        mult1 = np.zeros(width)
        mult2 = np.zeros(width)

        # 产生所有 (1,1,1) 的组合
        choice = []
        for i in range(width-1,-1,-1):
            subchoice = np.ones((len(newVars)))
            for j in range(len(newVars)):
                subchoice[len(newVars)-1-j] = (i >> j) % 2
            choice.append(subchoice)

        # 处理 mult1
        for index,value in enumerate(choice):
            # value: newvars 的取值组合
            newData[index,:-1] = np.array(value)
            bool_index = True
            for i,var in enumerate(newVars):
                if var in self.vars :
                    bool_index &= (self.data[var]==value[i]) # 选择 df 中
var = value[i] 的行(value = 0 /1 )
            mult1[index] = self.data[bool_index].iloc[0,-1]

        # 处理 mult2
        for index,value in enumerate(choice):
            # value: newvars 的取值组合
            bool_index=True
            for i,var in enumerate(newVars):
                if var in f.vars:
                    bool_index &= (f.data[var]==value[i]) #
            mult2[index] = f.data[bool_index].iloc[0,-1]

        for index in range(len(choice)):
            newData[index,-1] = mult1[index] * mult2[index]

        newVars.append("Value")
```

```
        newDf= pd.DataFrame(newData,columns=newVars)
        names_ = newDf.columns.tolist()[:-1]
        newDf[names_] = newDf[names_].astype(np.int)
        return Factor(newDf)
```

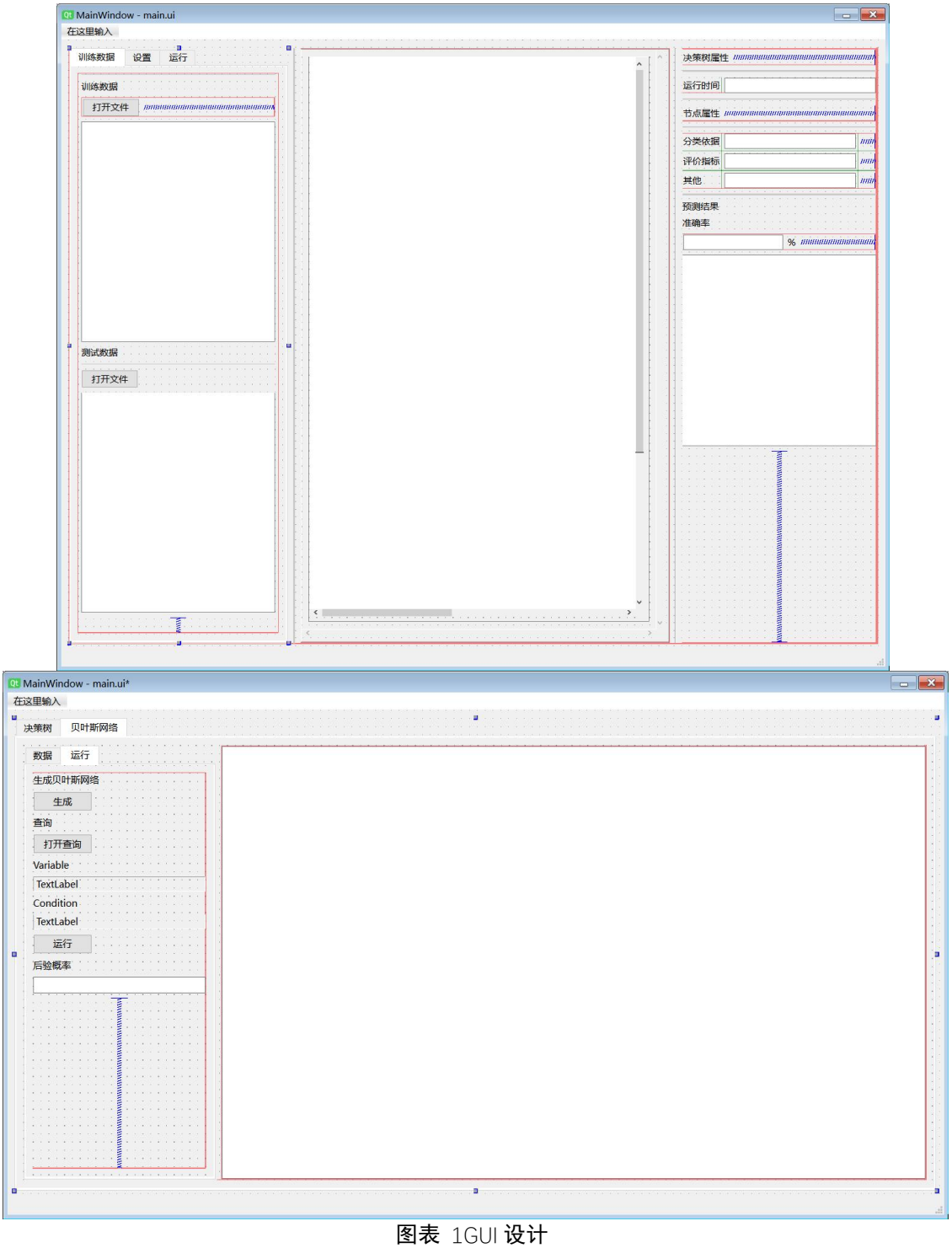
变量消元算法

变量消元

```
def ask(self,query:Query):
        orderedVar =  self.sortVars()
        qvars = list(query.vars.keys())
        qcond = list(query.conditions.keys())

        factor = None
        for index in orderedVar:
            newFactor = Factor()
            newFactor.fromNode(self.nodes[index],query.conditions)

            if factor is None:
                factor = newFactor
            else:
                factor = factor.mult(newFactor)
                # print("Mult:",end="")
                # factor.print()

            if self.names[index] not in qcond and self.names[index] not
in qvars:
                # 如果是隐藏变量，则消元
                factor = SumOut(self.names[index],factor)
            # print("Sum:",end="")
            # factor.print()
        # 归一化最后的 factor 结果
        # 假设 查询的 var 只有一个
        # TODO: 扩展到更高维
        result_df = factor.data
        arr = factor.data.to_numpy()[:,-1]  # 取出 prob
        if arr.shape[0] != 2:
            print("Wrong dimension!")
            return
        alpha = arr.sum()
        arr = arr/alpha
        var = result_df.columns.to_list()[0]
        pos = result_df[(result_df[var] ==
query.vars[var])].index.tolist()[0]
        return arr[pos]
```

2.3 图形化界面设计

本次项目中，主要使用 PySide6 框架实现 GUI 系统。整体 GUI 设计可以分为三个部分：数据输入，可视化显示和运行结果输出。整体 GUI 设计图如下所示。



图表 1GUI **设计**

如上图所示，设计图左侧部分为数据输入部分，可以打开 csv 文件，并且在训练数据的对应表格处显示训练数据。训练数据和测试数据分别在上下两个框中读入。

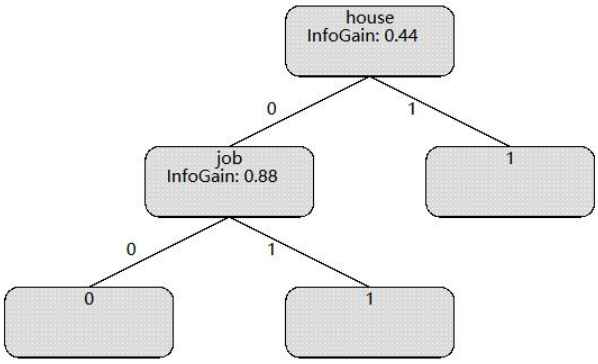
	age	job	house	credit	type
0	0	0	0	0	0
1	0	0	0	1	0
2	0	1	0	1	1
3	0	1	1	0	1
4	0	0	0	0	0
5	1	0	0	0	0
6	1	0	0	1	0
7	1	1	1	1	1
8	1	0	1	2	1
9	1	0	1	2	1
10	2	0	1	2	1

	age	job	house	credit	type
0	0	0	0	1	0
1	0	1	0	1	1
2	1	0	1	2	1
3	1	0	0	1	0
4	2	1	0	2	1
5	2	0	0	0	0
6	2	0	0	2	0

图形界面的第二个 tab 显示决策树的选择，有 ID3，C4.5,CART 三种可行的决策树

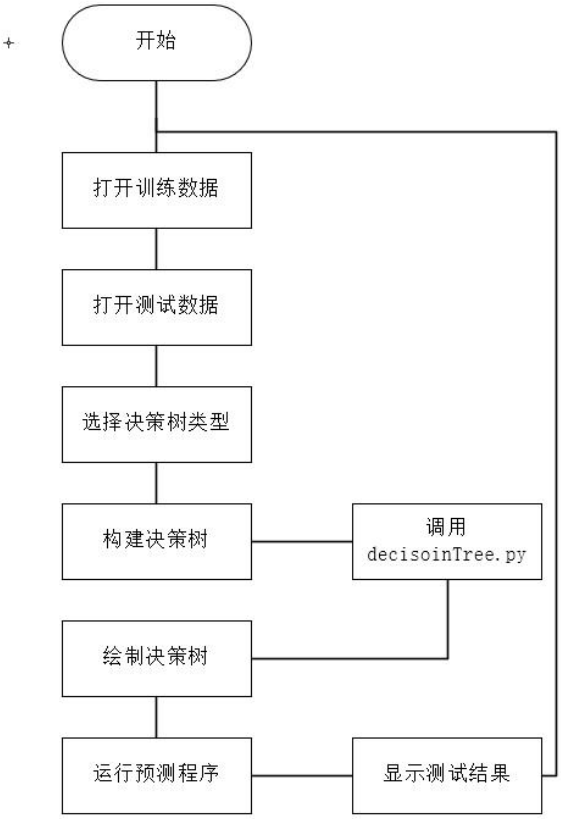
第三个 tab 是运行按钮。

在图形界面的中间，是决策树图形化的画布，在此区间中完成对决策树的图形化。



在图形界面的右侧显示决策树的建树细节、预测细节。

图形界面的运行逻辑是在初始化阶段声明图形界面的组件，然后在系统循环中循环绘制图形界面。图形界面从输入到输出的整体流程如下所示。



图表 2GUI 运行流程图

贝叶斯网络部分将一个文件夹中的所有文件读入，作为构造贝叶斯网络的成分，每一个结点对应一个 csv，文件，在文件中描述了不同变量之间的依赖关系。

B	E	A
1	1	0.95
1	0	0.94
0	1	0.29
0	0	0.001

图表 3 贝叶斯网络定义文件之一

贝叶斯网络通过一个 input.txt 文件作为 Query 查询输入，格式如下所示。

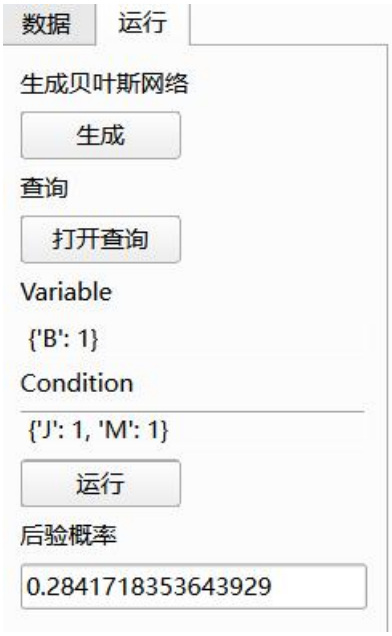
B
1
J,M
1,1

打开文件夹后，左边数据列表显示文件夹中的所有文件。单击文件名称可以查看其内容。

图表 4 文件夹打开界面

在“运行”的 tab 下，点击按钮打开 query 输入文件，得到文件内容，显示在两个 label 中。

点击运行按钮，得到后验概率值



3 实验过程

左图显示了读入训练数据后的表格视图。

3.1 环境说明

操作系统：Windows 10

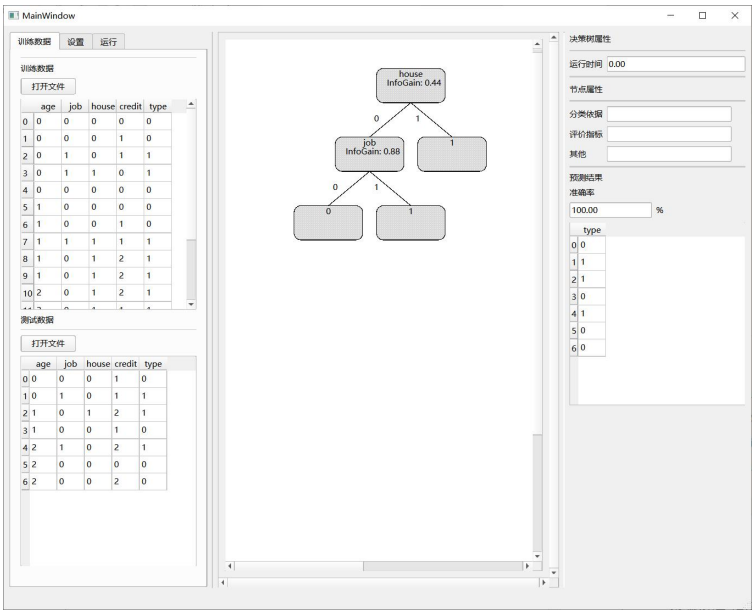
开发语言：Python

开发环境：Python >= 3.0.0

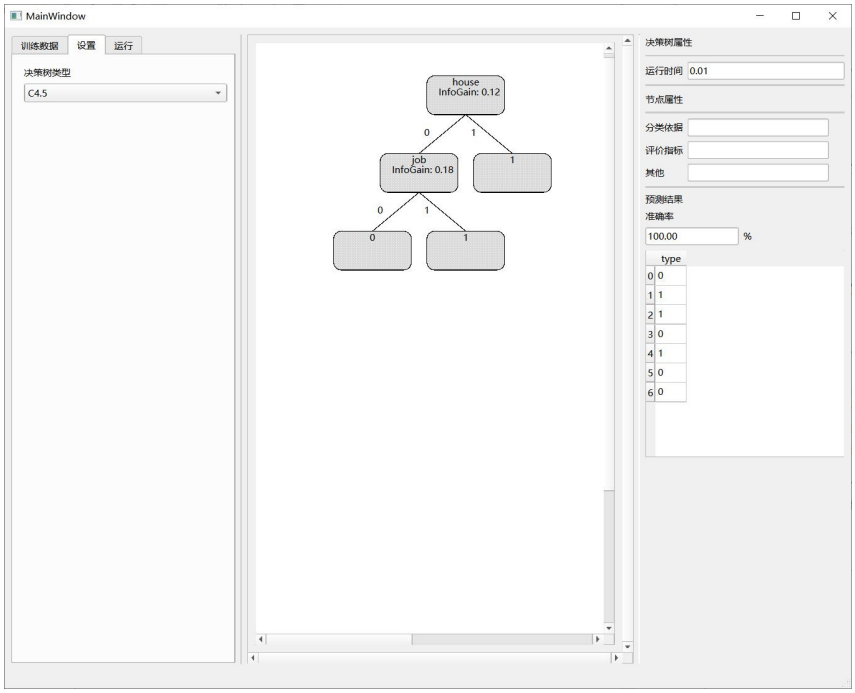
3.2 源代码文件清单

文件夹	文件名	功能
.	Main.ui	Ui 设计
.	Mainwindow_ui.py	Ui 设计对应 python 文件
.	Mainwindow.py	主文件
BN	*	贝叶斯网络相关文件
DecisionTree	*	决策树相关文件
Module	*	Ui 相关模块
Netstruct	*	贝叶斯网络定义
Query	*	贝叶斯网络查询

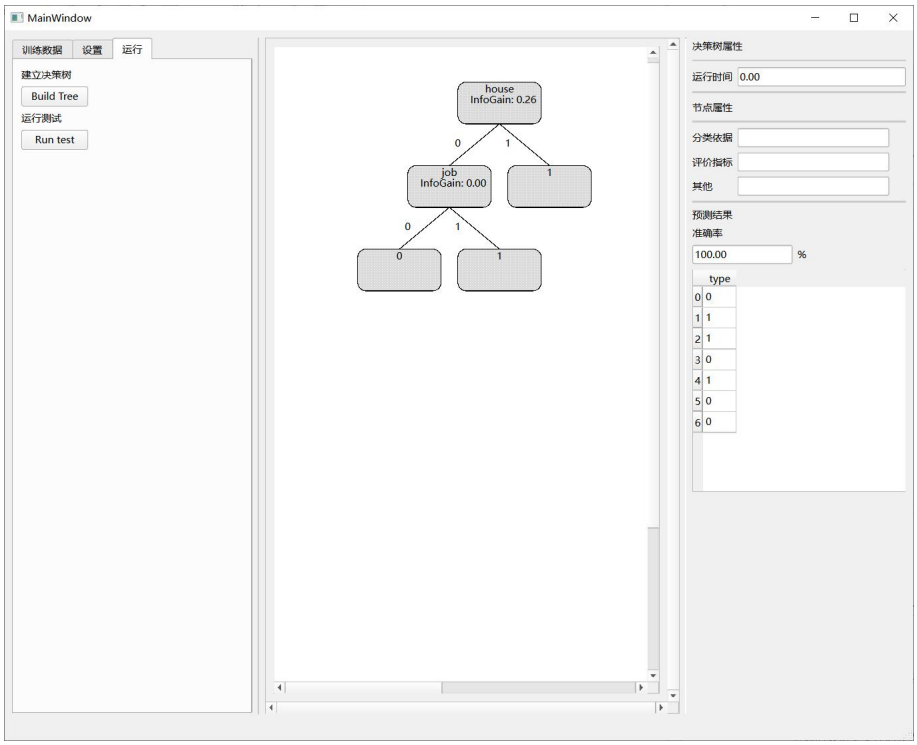
3.3 实验结果展示



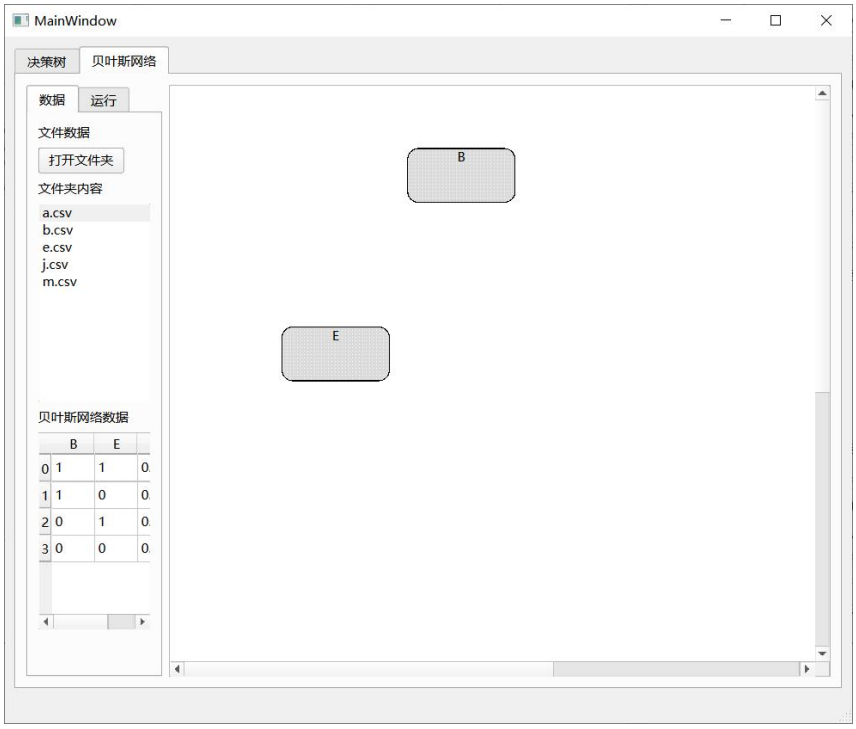
图表 5 ID3 运行结果



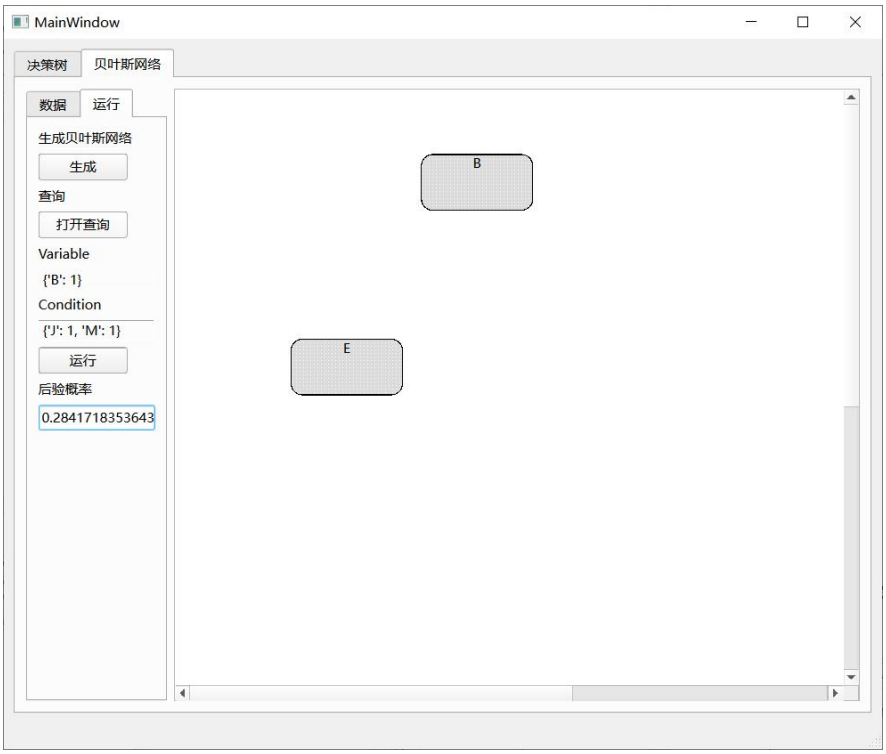
图表 6 C4.5 运行结果



图表 7CART 运行结果



图表 8 贝叶斯网络运行结果



图表 9 贝叶斯网络运行结果

4 总结

4.1 实验中存在的问题及解决方案

（1）如何有机统一不同特征的不同的取值可能和在推理时的便利实现相结合：Python 内建的 dict 字典类型是一个很好的选择，通过设置特征的不同的可能取值为 key 值，对应的 value 为递归返回的结果，若连接的为叶节点，则 value 为 '0' 或 '1' 表示正负例，若为决策树的内部节点，则储存对应的对象，将查询请求递归处理直到叶节点。

（2）提升程序的适用性，使得前处理环节简单：程序使用 Python 原生的 csv 模块进行 CSV 格式的数据集文件的读入，提升程序的可用性，不需要人工再对分隔符、换行符等做限制。

（3）实验中，QgraphicsView 中使用 Qpainter 调用 drawText 方法时，不能使用\n 转义符输出多行语句，会导致闪退，并且没有报错。输出多行信息需要调用多次 drawText 方法。

（4）自定义 QgraphicsItem 时，需要定义 BoundRect 方法，否则不能在 graphics view 中绘制，同样导致运行时错误，且不会产生报错信息。

4.2 心得体会

在本次实验中，我们小组成员基于分工协作，完成了基于决策树对目标进行分类的程序和其相应的图形化界面显示。通过本次实验，我们动手实践了决策树算法的编写，加深了对决策树算法的知识，了解到了不同决策树算法之间的差异以及决策树的优化方式。

在本次实验中，我们使用 python 编写图形化界面，第一次了解如何从用户的角度完善页面，从使用者而非是程序编写者的角度看待功能。为此，我们设计了画布、多种选择栏和输入栏，以及信息显示栏，为了达成更好的交互体验。

在本次试验中，我们更是了解到沟通协作在团队工作中的重要性，通过组员的积极配合和及时沟通，我们顺利完成本次作业。

4.3 后续改进方向

（1）使决策树能学习连续属性，目前只实现了基于离散属性来生成决策树。

（2）现实任务中常会遇到不完整样本，需要使决策树能利用有缺失属性值的训练样例进行学习。

（3）进一步学习多变量决策树，掌握每个非叶结点不仅考虑一个属性的决策树算法。

（4）使用更加灵活的树布局算法，防止显示不清的问题。

4.4 总结

在本次实验中，我们小组成员基于分工协作，完成了基于决策树对目标进行分类的程序和其相应的图形化界面显示。通过本次实验，我们动手实践了决策树算法的编写，加深了对决策树算法的知识，了解到了不同决策树算法之间的差异以及决策树的优化方式。

在本次实验中，我们使用 python 编写图形化界面，第一次了解如何从用户的角度完善页面，从使用者而非是程序编写者的角度看待功能。为此，我们设计了画布、多种选择栏和输入栏，以

及信息显示栏，为了达成更好的交互体验。

在本次试验中，我们更是了解到沟通协作在团队工作中的重要性，通过组员的积极配合和及时沟通，我们顺利完成本次作业。

5 附录

5.1 成员分工与自评

朱昀玮：编写 PySide Gui 程序，编写贝叶斯网络算法。10/10

林日中：决策树算法编写及优化，10/10

陆明奇：决策树算法编写及优化，10/10